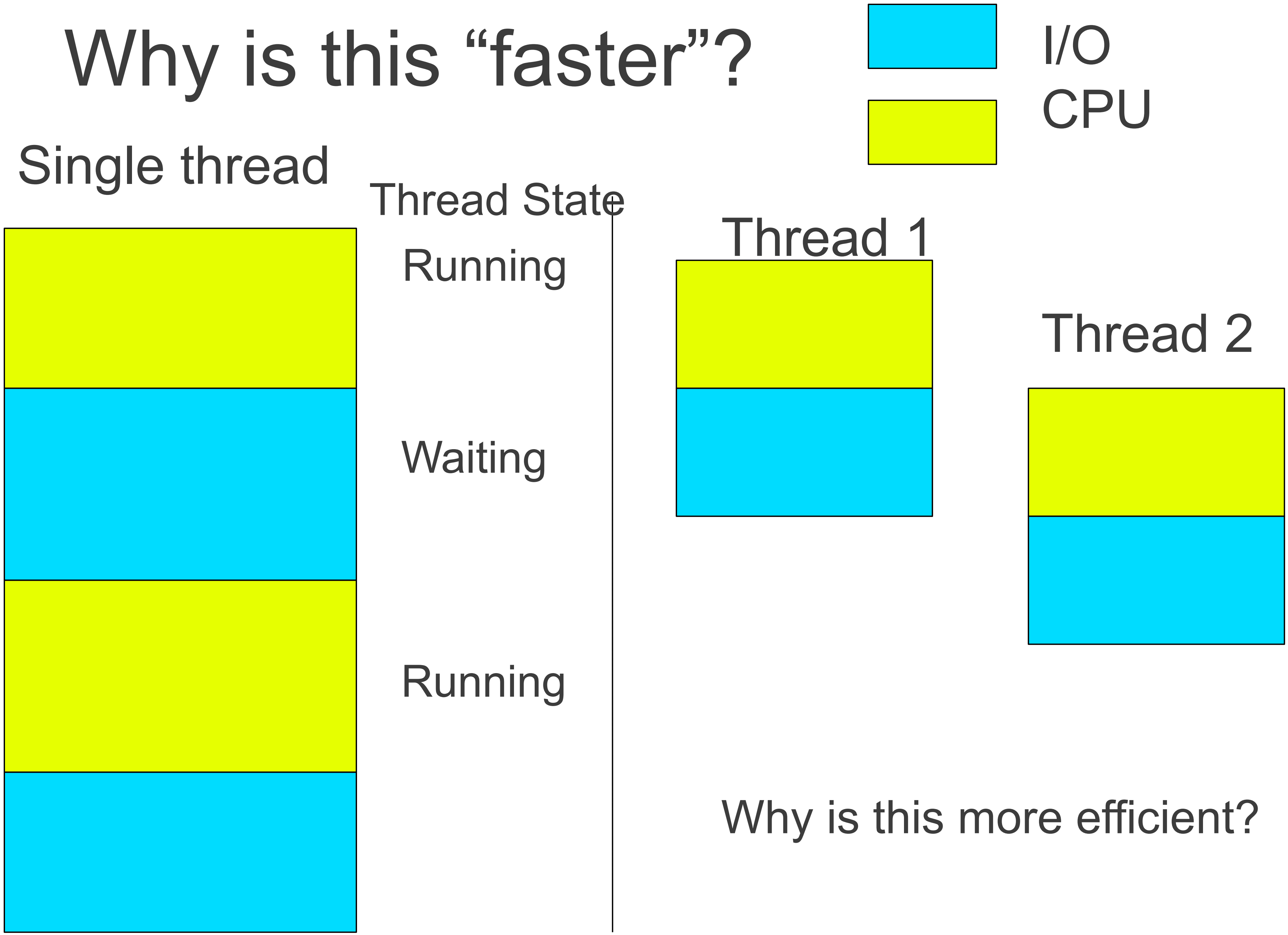
Section 4
Processes, kernel threads, user threads, locks

Why use threads?

- Perform multiple tasks at once (reading and writing, computing and receiving input)
- Take advantage of multiple CPUs
- More efficiently use resources

Why is this "faster"?



Quick view

- Process
  - Isolated with its own virtual address space
  - Contains process data like file handles
  - Lots of overhead
  - Every process has AT LEAST one kernel thread
- Kernel threads
  - Shared virtual address space
  - Contains running state data
  - Less overhead
  - From the OS's point of view, this is what is scheduled to run on a CPU
- User threads
  - Shared virtual address space, contains running state data
  - Kernel unaware
  - Even less overhead

Trade-offs

- Processes
  - Secure and isolated
  - Kernel aware
  - Creating a new process (address space!) brings lots of overhead
- Kernel threads
  - No need to create a new address space
  - No need to change address space in context switch
  - Kernel aware
  - Still need to enter kernel to context switch
- User threads
  - No new address space, no need to change address space
  - No need to enter kernel to switch
  - Kernel is unaware. No multiprocessing. I/O blocks all user threads.

Implicit overheads

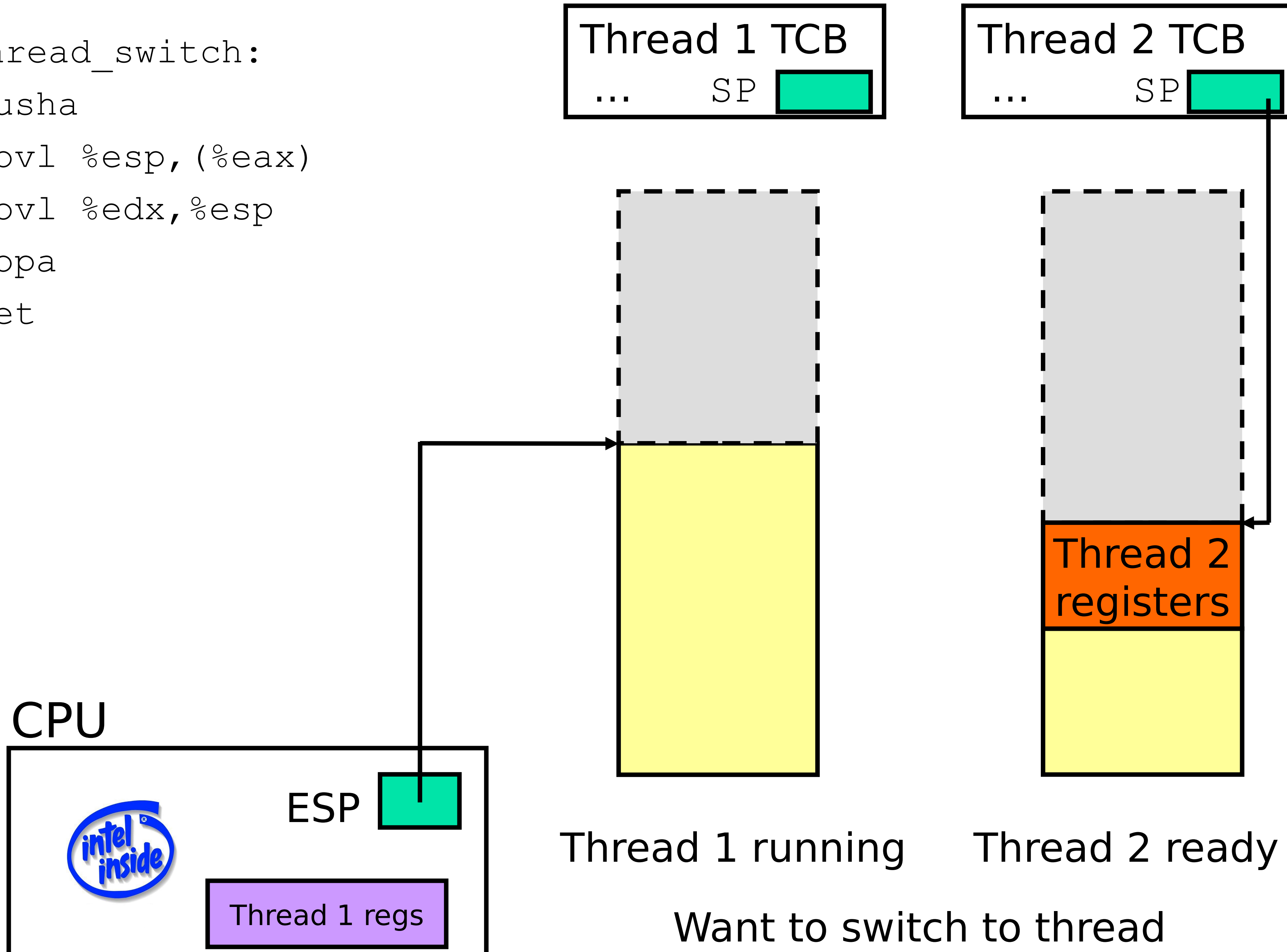
- Context switching between processes is very expensive because it changes the address space.
  - But changing the address space is simply a register change in the CPU?
  - But it requires flushing the Translation Look-aside Buffer.
- Context switching between threads has a similar overhead. Suddenly the cache will miss a lot.

When should I use which?

- Process
  - When isolation is necessary
    - Like in Chrome
- Kernel threads
  - Multiprocessor
  - heavy CPU per context switch
  - Blocking I/O
  - Compiling Linux
- User threads
  - Single processor or single kernel thread
  - Light CPU per context switch
  - Little or no blocking I/O

Context switching

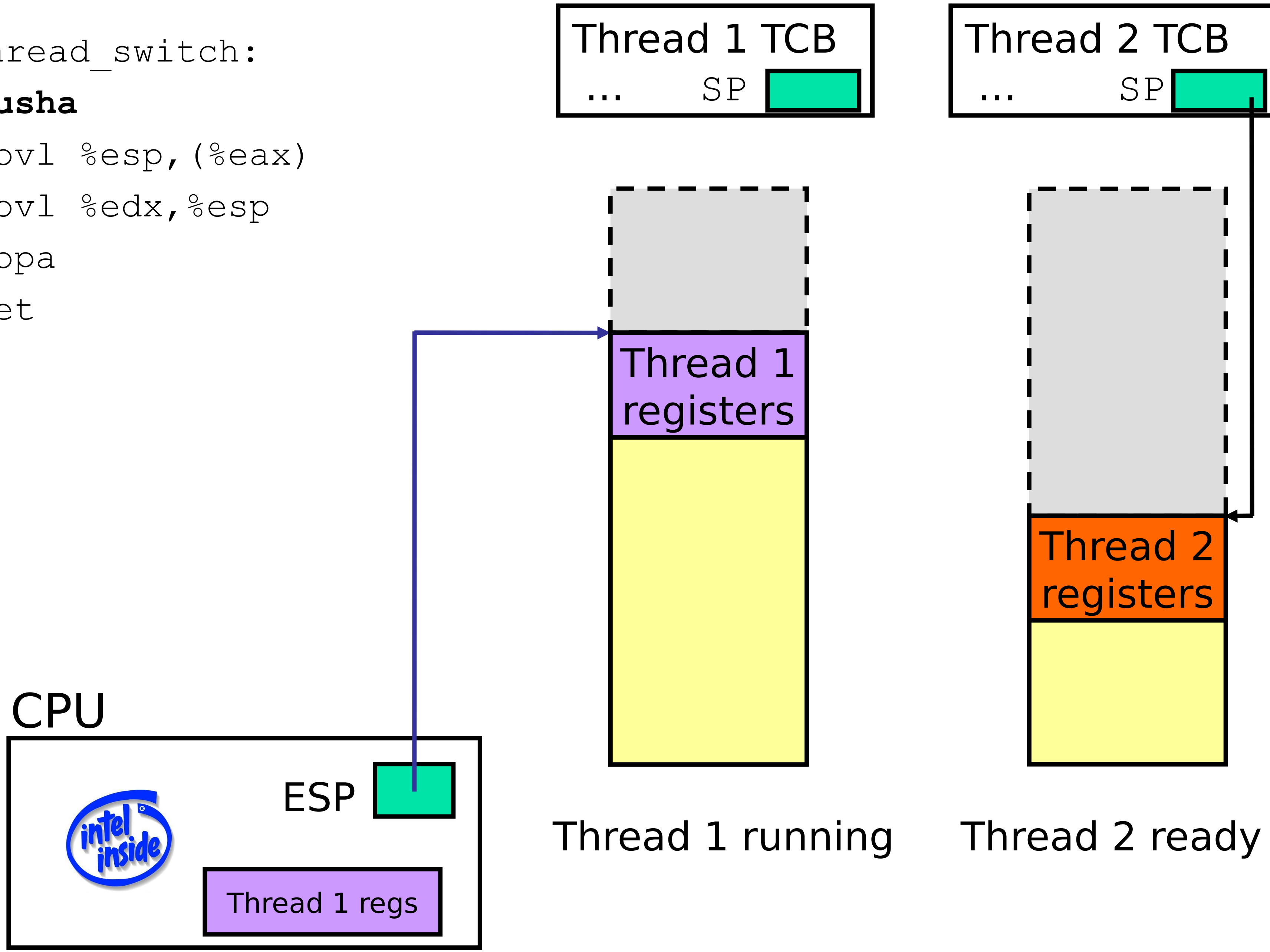
```
Xsthread_switch:
   pusha
   movl  %esp,(%eax)
   movl  %edx,%esp
   popa
   ret
```



Want to switch to thread 2…

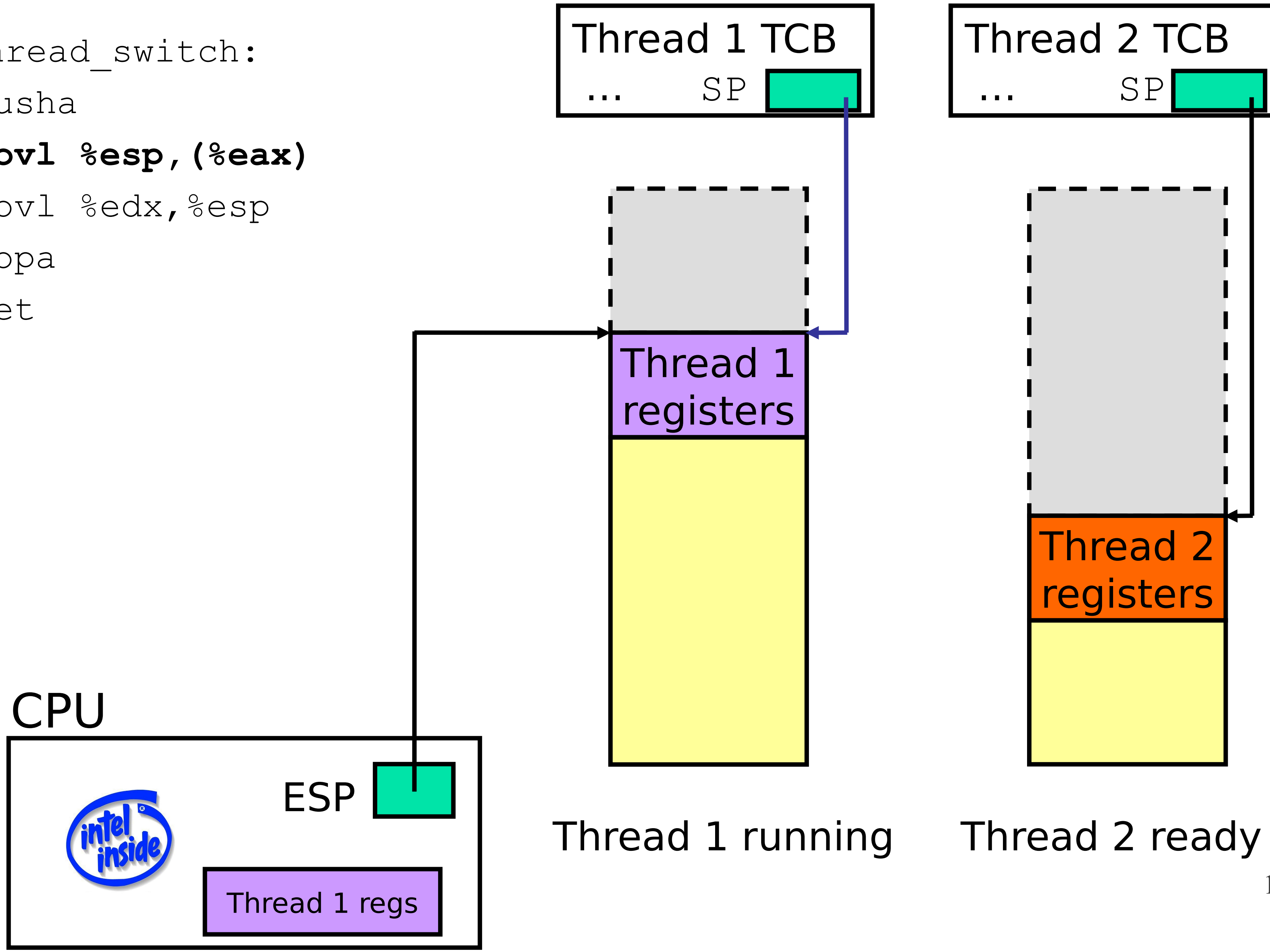
Push old context

```
Xsthread_switch:
    pusha
    movl %esp,(%eax)
    movl %edx,%esp
    popa
    ret
```



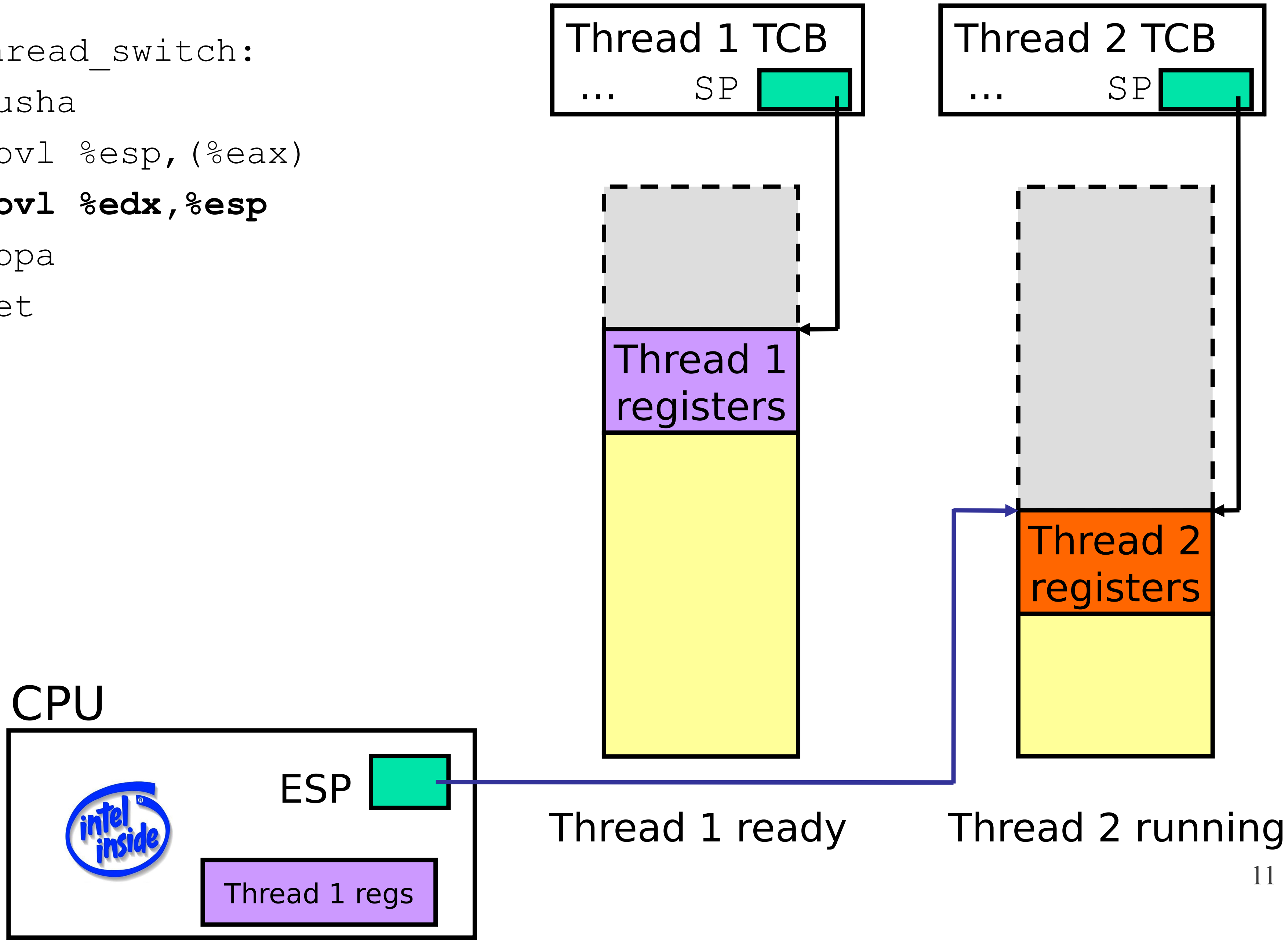
Save old stack pointer

```
Xsthread_switch:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```



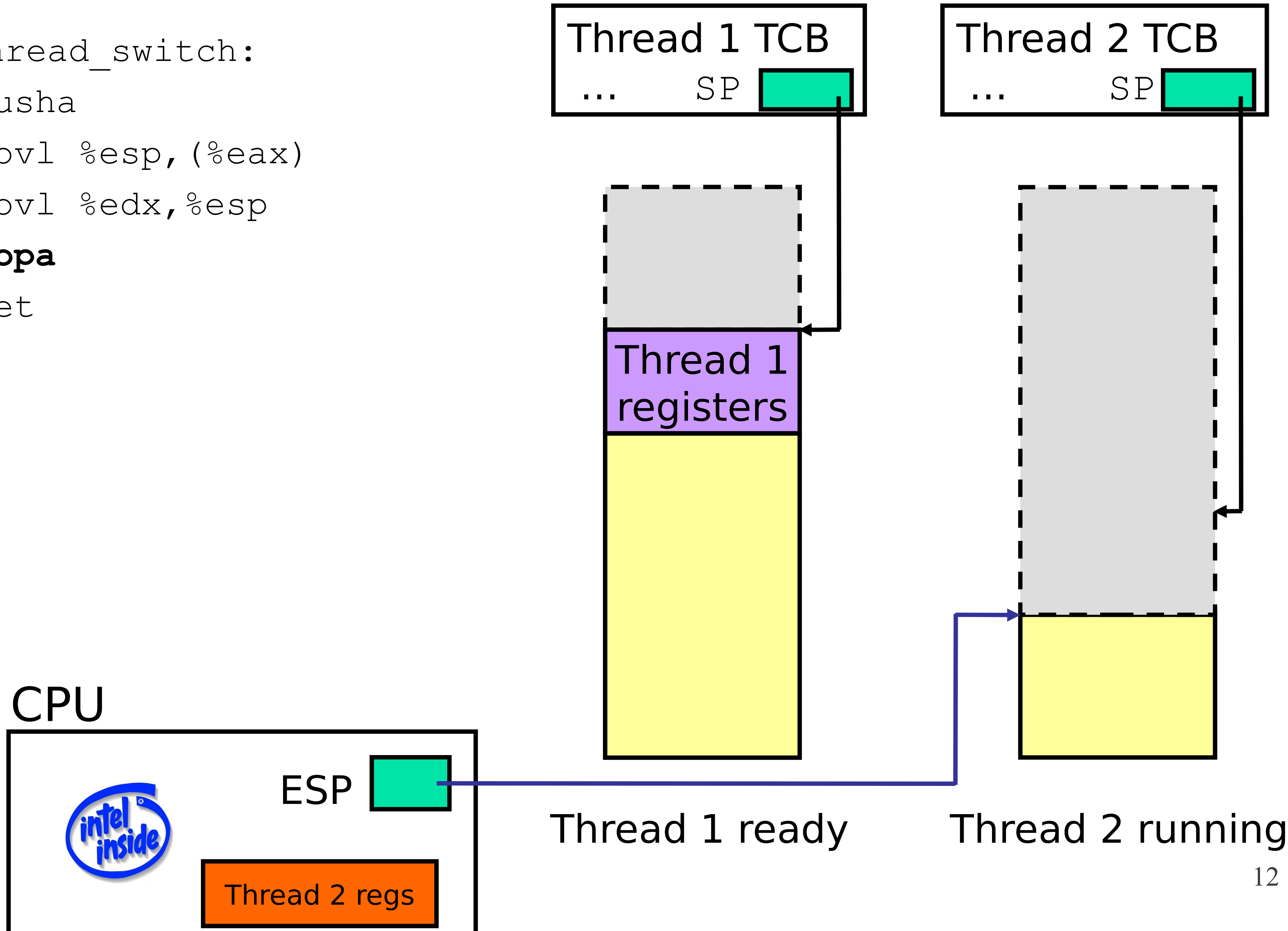
Change stack pointers

```
Xsthread_switch:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```



Pop off new context

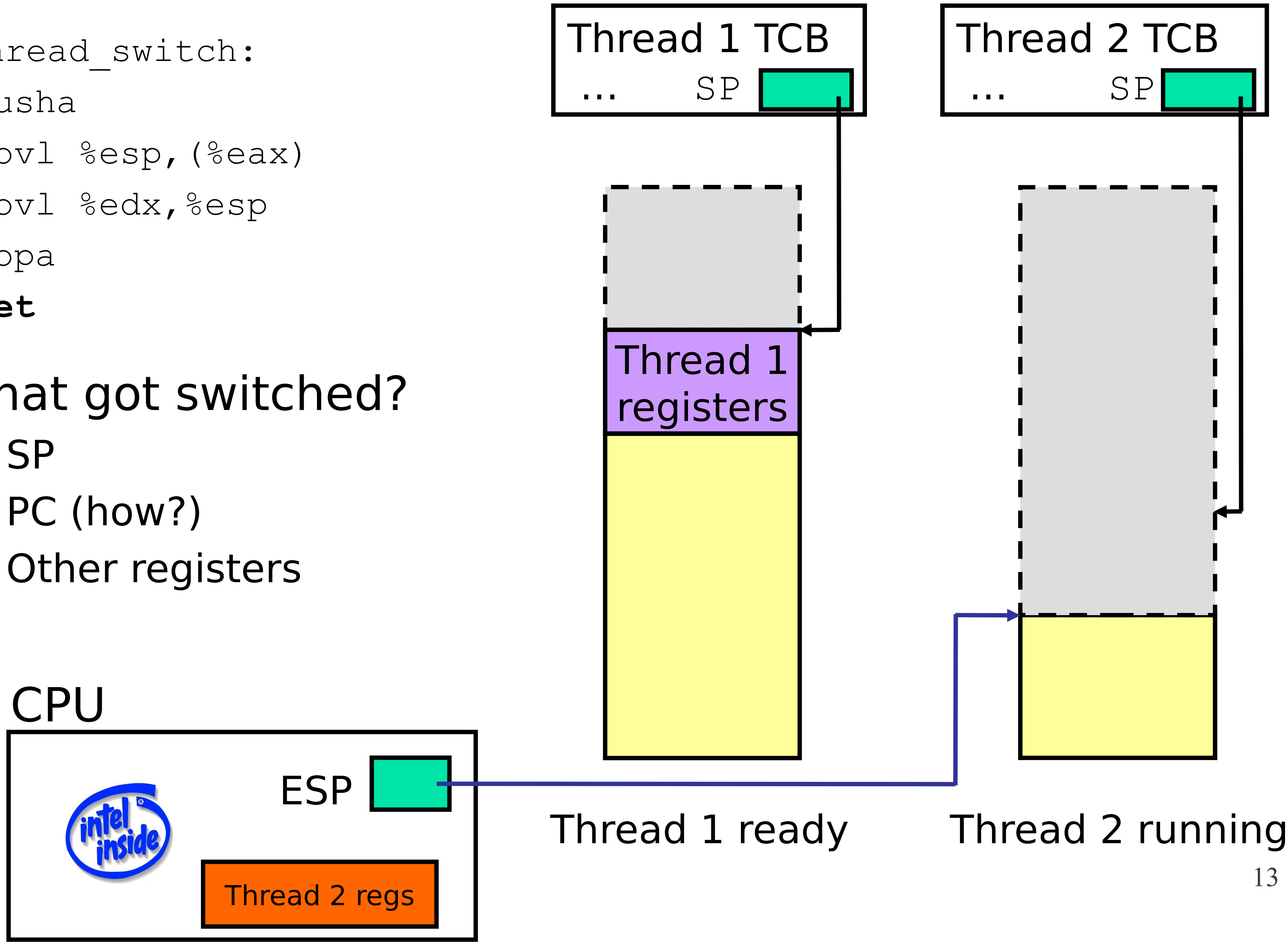
```
Xsthread_switch:
   pusha
   movl %esp,(%eax)
   movl %edx,%esp
   popa
   ret
```



Done; return

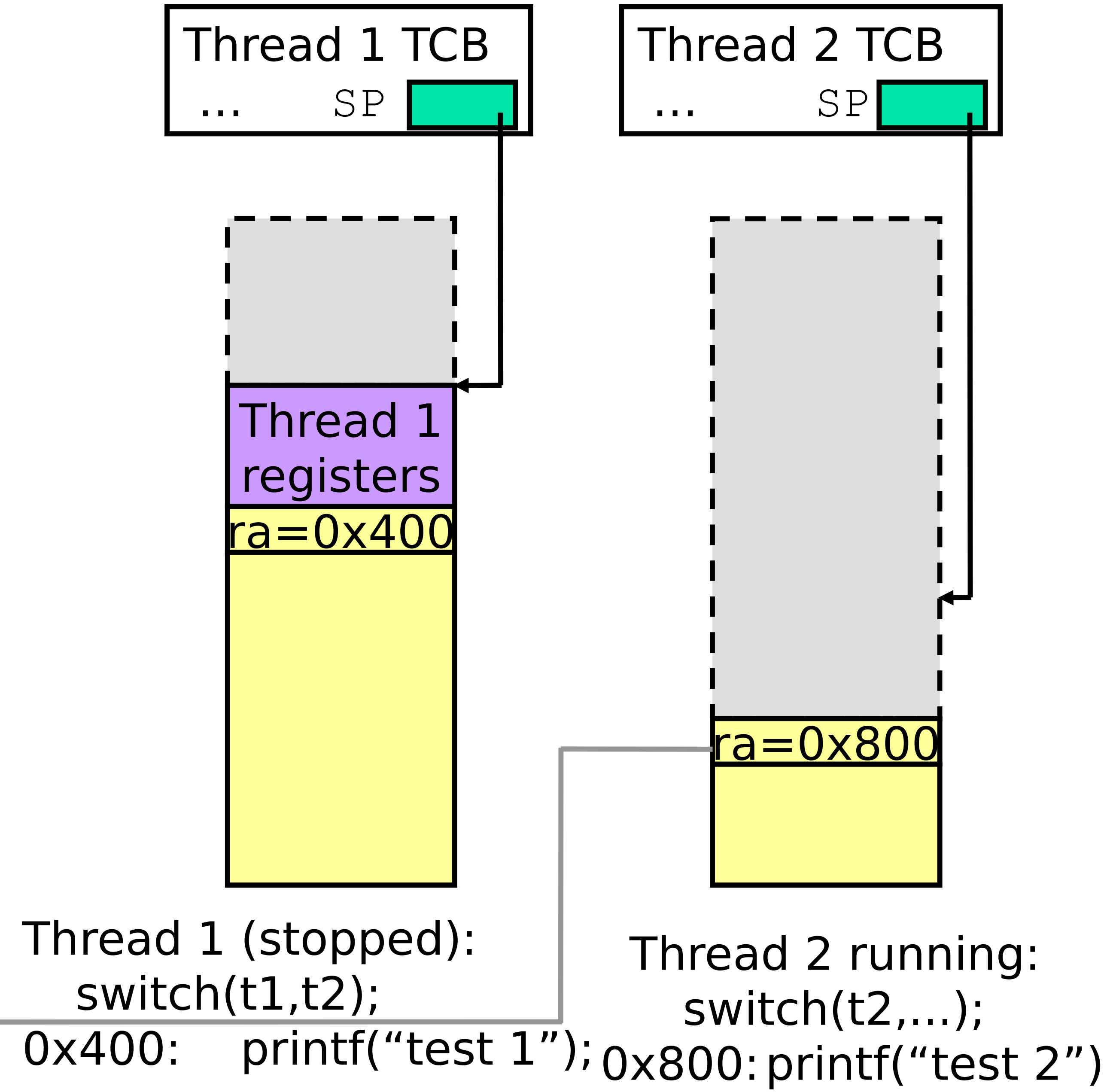
```
Xsthread_switch:
    pusha
    movl %esp,(%eax)
    movl %edx,%esp
    popa
    ret
```

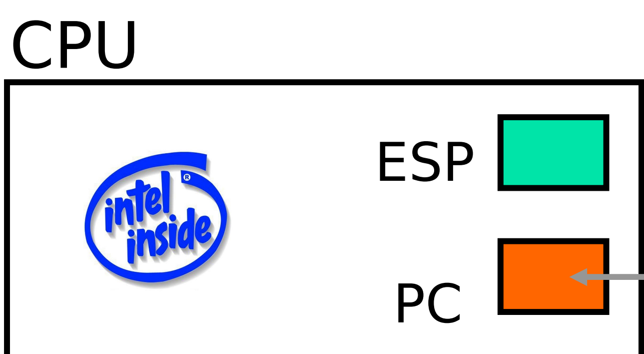
- What got switched?
  - SP
  - PC (how?)
  - Other registers



Adjusting the PC

- **ret** pops off the new return address!





Context Switching

- So was this for kernel threads or user threads?
  - Trick question! This can be accomplished in either kernel or user mode.

Theading Models

Between kernel and user threads, a process might use one of three models:

- One to one (1:1)
  - Only use kernel threads without user level threads on top of them.
- Many to one (M:1)
  - Use only one kernel thread with many user level threads built on top of them.
- Many to Many (N:M)
  - Use many kernel threads with many user level threads.

Threading Models

- Many to many sounds nice, intuitively but...
    - ...it can actually get problematic in its complexity
    - See Scheduler Activations

- Linux actually runs One to one
- Windows runs a lazy version of Scheduler Activations.

Linux and threads/processes

- You must have noticed in your project you deal with a Linux structure called a “task_struct”. Is this a PCB or TCB?

task_structs

- Linux has no explicit concept of a “thread” (or a process) but “tasks”.
- A task is a “context of execution” or COEs.
  - COEs can share anything, nothing, or something in-between.
- This allows for more capabilities like:
  - An external “cd” program. (shares fs struct and cwd).
  - “external IO daemons”. (shares file descriptors)
  - vfork (shares address space).
- Linus' argument for this paradigm: http://www.evanjones.ca/software/threading-linus-msg.html

Locks

- If you need to protect shared data and critical sections, you need some primitive to work with.
- But, there are lots of design choices in locking and synchronization.

Spinning vs Blocking

- Spinning
  - If the lock is not free, repeatedly try to acquire the lock.
- Blocking
  - If the lock is not free, add the thread to the lock's wait queue and context switch.
- When to use which?
  - Spinning is good for small critical sections.
  - Also good on multiprocessors.
  - If the overhead of the context switch is less than the time spent waiting (spinning), then blocking is preferable.
    - But remember the implicit overhead of context switching as well.
  - Spin locks are good for fine-grained work like you might see in your OS.
  - Blocking is good for coarse-grained work like protecting large data structures.

Pessimistic Vs Optimistic Locking

- Pessimistic locking checks a lock before updating or entering a critical section.
  - This commonly uses test_and_set.
  - This ensures that the current thread is the only one operating.
- Optimistic locking checks that an update will not break the structure.
  - It does this by reading an initial value then checking that this value hasn't changed with compare_and_swap.
  - If the value has changed, abort and try again.
  - Therefore, any number of threads might be operating on a "critical section."

When to use which?

- “Make the common case fast.”
- Pessimistic locking assumes that the common case is contention.
  - We won't waste time trying to run through critical section if we only end up aborting.
  - An OS has lots of small, commonly used data structures and critical sections.
- Inversely, optimistic locking assumes that most of the time there isn't contention.
  - Optimistic locking is like database transactions. They assume the will not commonly abort.
  - Also good when data is commonly read but rarely written.
  -

Granularity of locks

- One big lock.
  - Low overhead.
  - Fewer memory references.
  - Less concurrency.
- Many little locks.
  - Higher overhead.
  - More memory references. Greater capacity for bus contention and cache storms.
  - Greater concurrency.
- Avoiding locks entirely...?

Moral of the story...?

- Know thy workload.
  - Generally these are statically decided design choices.

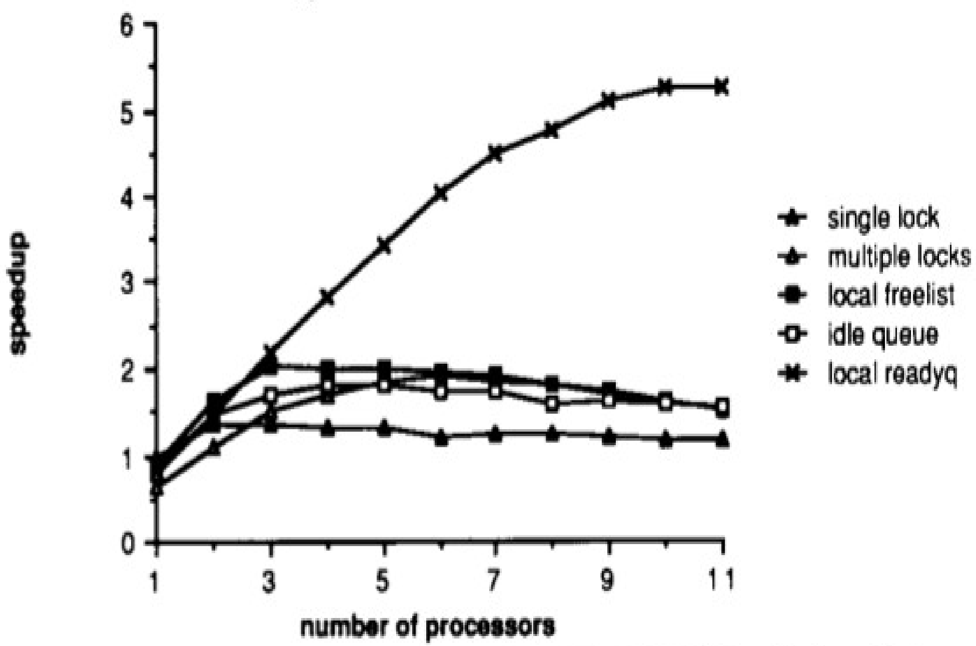


Fig. 2. Speedup to create, start, and finish 1 000 000 null threads (measured).

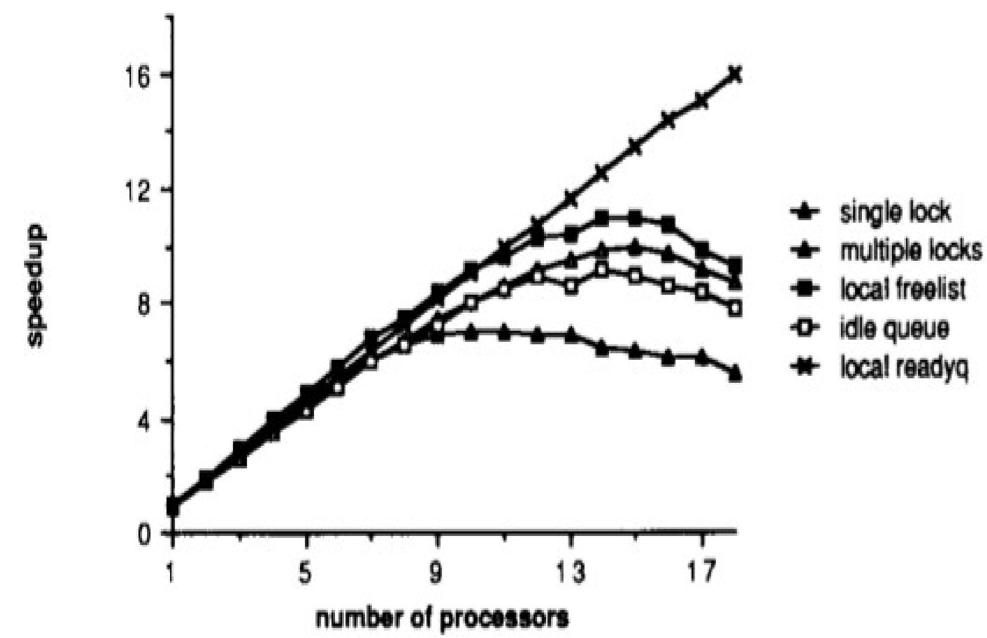


Fig. 3. Speedup, user work = 300 μs (measured).